Physical Memory Structures

Information for computing systems is stored in mechanisms of the following type:

1. Random Access Memory (RAM)
   - read-write memory (RWM)
   - read-only memory (ROM)
2. Content Addressable Memory (CAM) or Associative Memory (AM)
3. Sequential Access Memory (SAM)
4. Direct Access Memory (DAM)

RAM

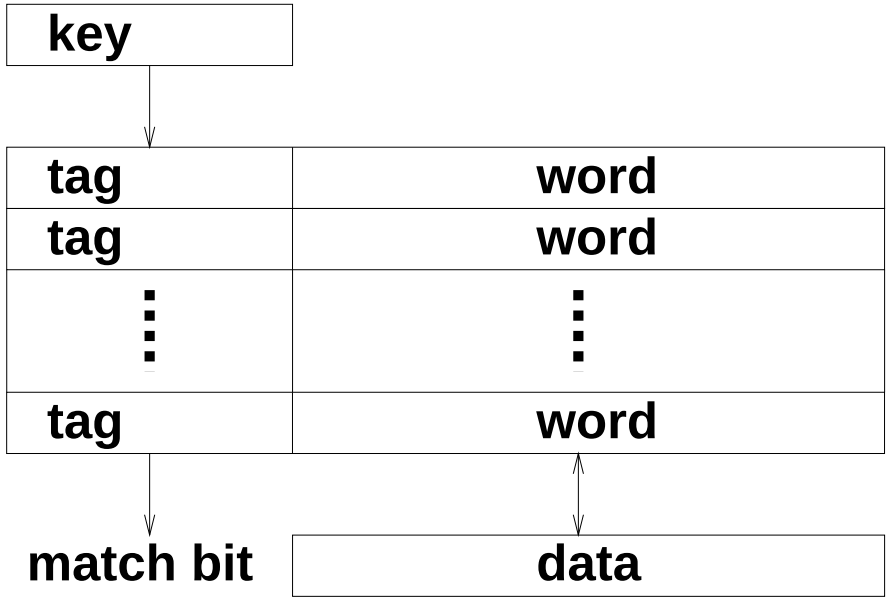
Data is stored and written directly from a given address. Regardless of the location of the data, all accesses require the same amount of time.

Read-write memory (RWM): Data can be read or written to/from any location. Data is usually stored in terms of words. A RWM with 2^n words and m-bits per word has two critical registers: an n-bit memory address register (MAR) and an m-bit memory data register (MDR).

Read-only memory (ROM): Data can only be read from each location. Data is usually fixed in the words of a ROM either by the manufacturer or by some off-line manner.

AM: Associative Memory

Associative Memory (AM): Data is read or written into memory locations based upon a tag field associated with the data.



SAM: Sequential Access Memory

Sequential Access Memory (SAM): Data is read/written in a sequential manner. That is, as each new datum to be read/written, it is stored after the previously read/written datum. Typically, the data is stored on an extended medium such as tape that is passed over a data read/write head.

DAM: Direct Access Memory

Direct Access Memory (DAM): A combination of random-access and sequential-access memory. Data is stored on a rotating disk or drum. The disk (drum) is organized into tracks onto data is sequentially stored. The read/write heads are then positioned (random-access) over the appropriate track and the data is then read/written (sequentially) a the appropriate place on the rotating disk.

Interleaved Memories

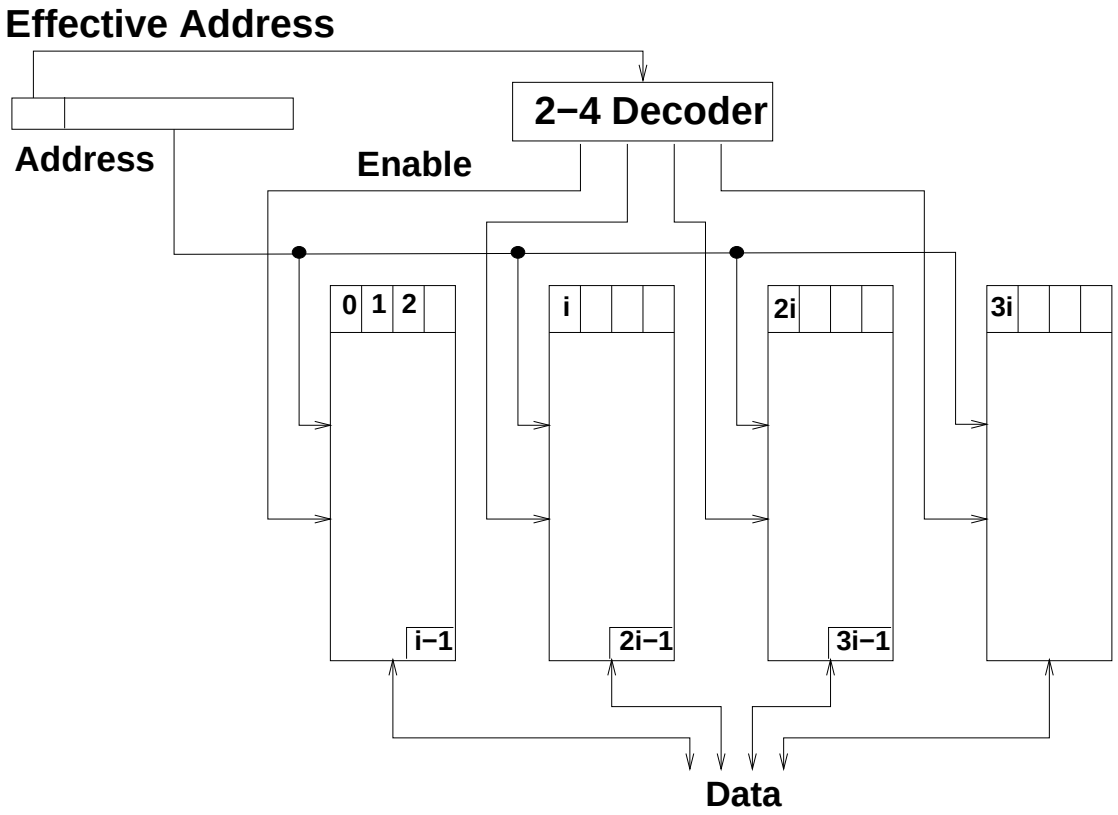
A use of several smaller memory modules that collectively form larger memories.

- **2 types:** High-Order Interleaving
  Low-Order Interleaving

- Can be and are used together:
  - each SIMM/DIMM is composed by Low-Order interleaving
  - motherboard memory banks constitute High-Order interleaving

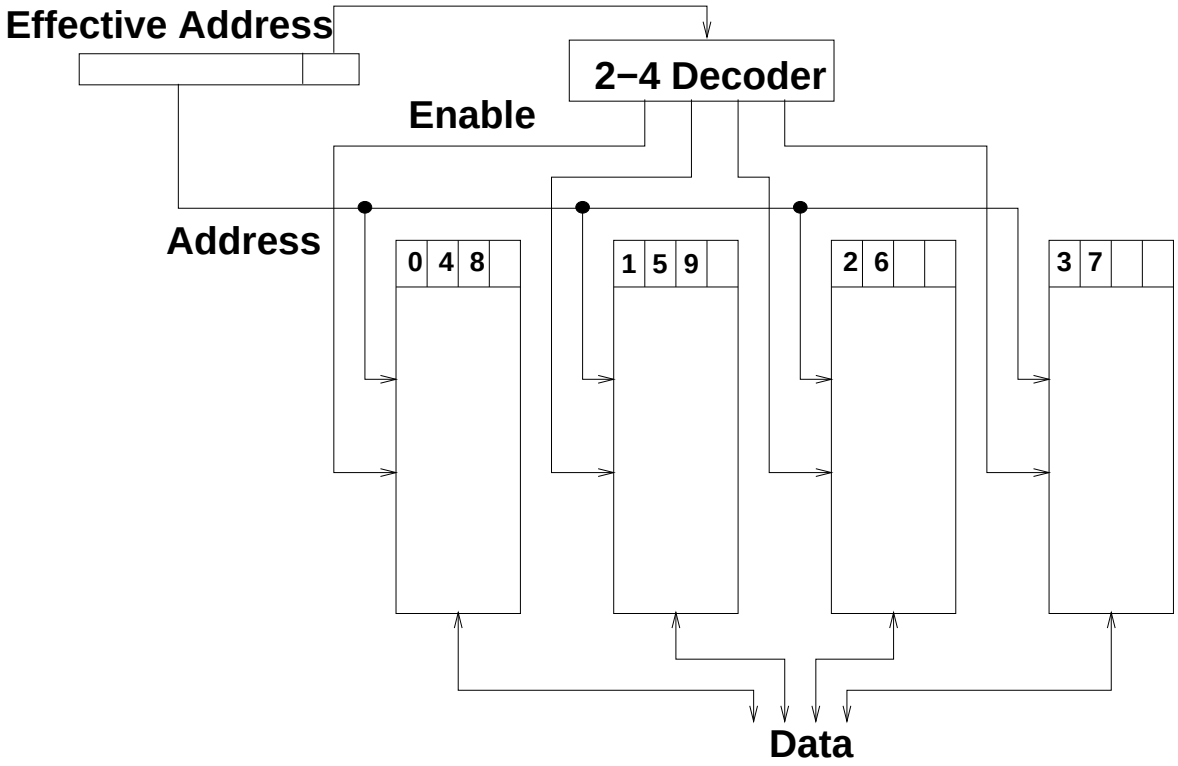
High-Order Interleaving

Distribute addresses in $M = 2^m$ modules so that each module i ($0 \leq i < M$) contains consecutive addresses $i2^{n-m}$ to $(i+1)2^{n-m} - 1$. High order address bits enable the memory module. Low order bits select the word in the module.



Low-Order Interleaving

Distribute the addresses so that consecutive addresses are located within consecutive modules. Low order address bits enable the memory module. High order bits select the word in the module. Modules easily combined to build wider data widths (all modules are enabled on each R/W).

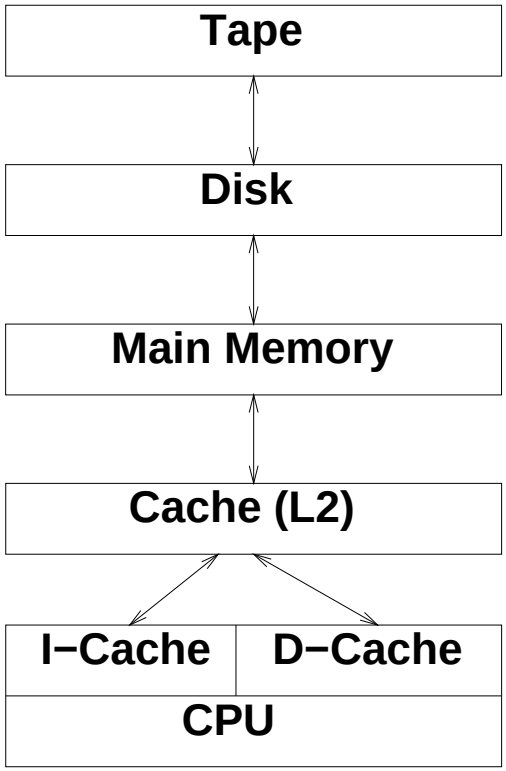


The Memory Hierarchy

Basic Problem: How to design a reasonable cost memory system that can deliver data at speeds close to the CPU's consumption rate.

Current Answer: Construct a memory hierarchy with slow (inexpensive, large size) components at higher levels and with fastest (most expensive, smallest) components at the lowest level.

Migration: As it is referenced, migrate data into and out-of the lowest level memories.



How does this help?

- Programs are well behaved and tend to follow the observed "principles of locality" for programs.

  Principle of temporal locality: states that a referenced data object will likely be referenced again in the near future.

  Principle of spatial locality: states that if data at location x is referenced then it is likely that a nearby location ($x + \Delta x$) will be referenced in the near future.

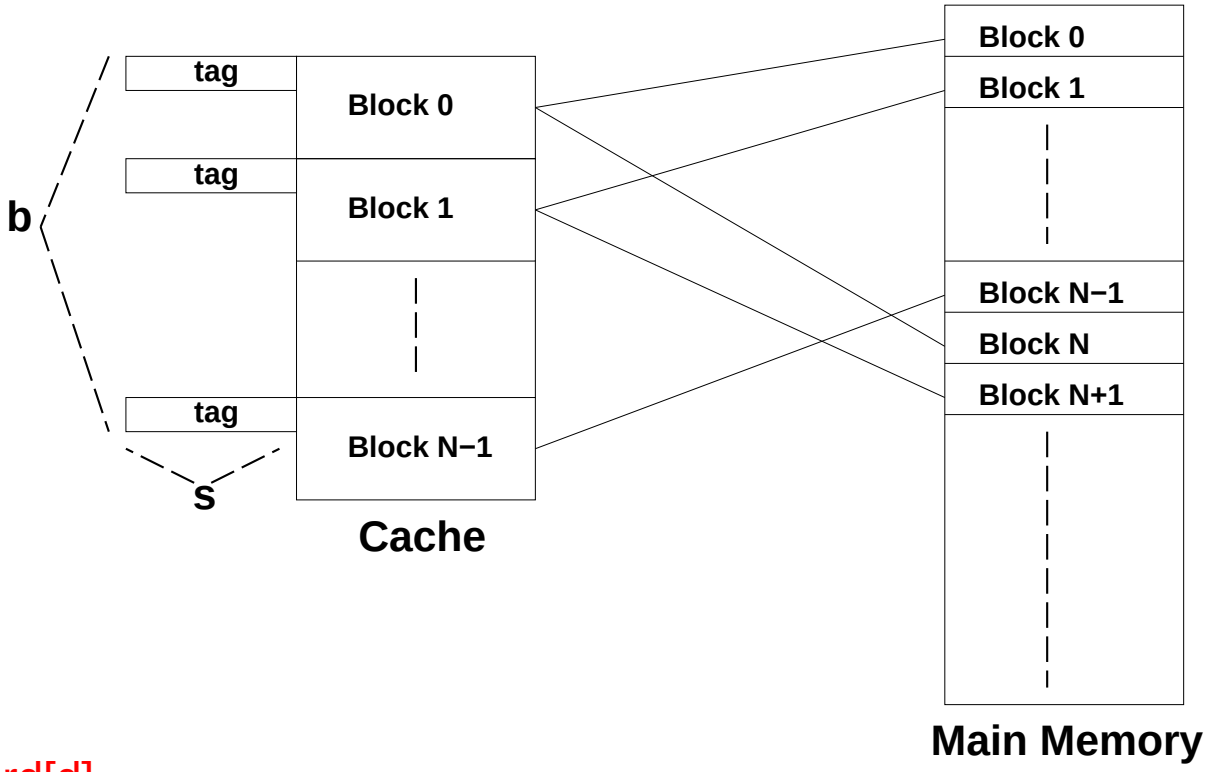
- Also consider the *90/10 rule*: A program executes about 90% of its instructions from about 10% of its code space.

Cache Memory Overview & Structures

- Similar to paged virtual memory w/ fixed-sized blocks mapped to the cache (from next higher level memory).
- Due to speed considerations, all operation implemented in hardware.
- 4 types (mapping policies)
  - direct mapped
  - fully associative
  - set associative
  - sector mapped (not discussed further)

Direct Mapped

If cache is partitioned into N blocks then cache block k will contain only memory blocks k+nN (n = 0,1,...). Memory addresses are formed as (s, b, d) where s is the memory tag (n), b is the cache block (k) and d is the word in the block.

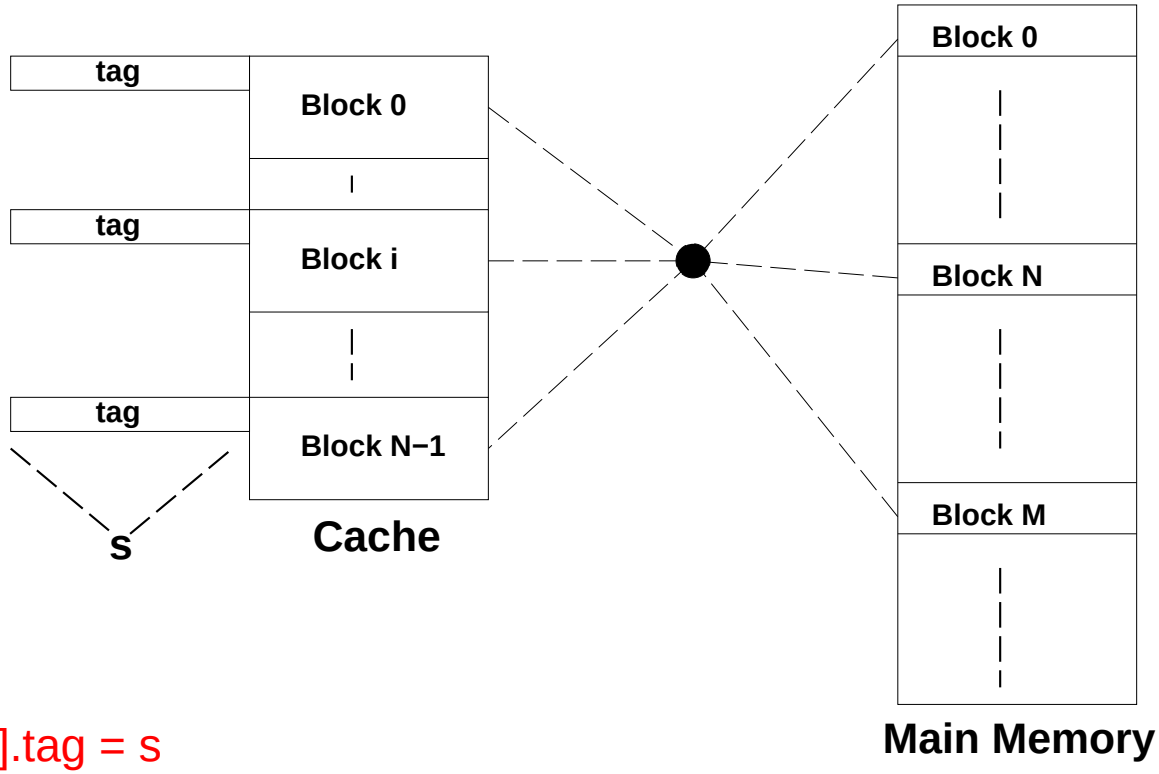


Lookup Algorithm

if cache[b].tag = s
  then return cache[b].word[d]
  else cache-miss

Fully Associative

Any block of MM can be mapped into any cache block. Memory addresses are formed as (s, d) where s is the memory tag, and d is the word in the block.



Lookup Algorithm

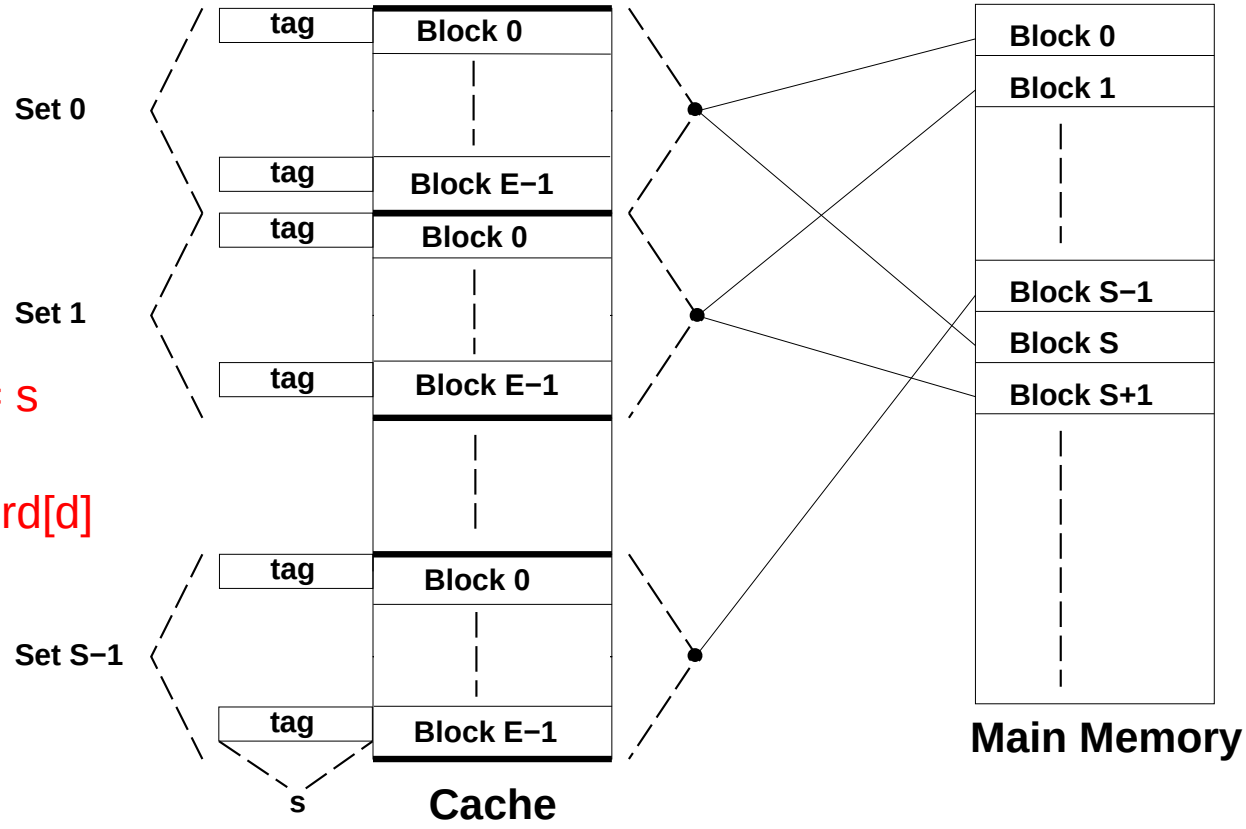
if $\exists\ k : 0 \leq k < N \wedge \text{cache}[k].\text{tag} = s$
  then return cache[k].word[d]
  else cache-miss

Set Associative

Compromise between direct and fully associative caches. Basic idea: divide cache into S sets with E = N/S block frames per set (N total blocks in cache). Memory addresses are formed as (s, b, d) where s is the memory tag, b is the cache set pointer, and d is the word in the block.

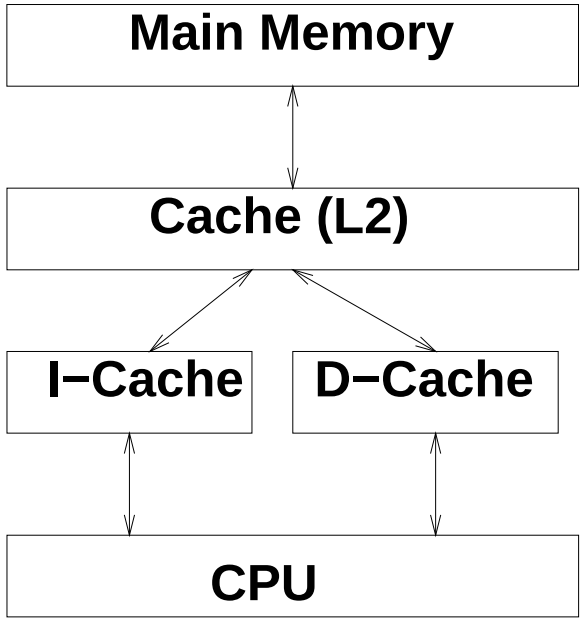
Lookup Algorithm

if $\exists\ k : 0 \leq k < E \wedge$
cache[b].block[k].tag = s
then return
cache[b].block[k].word[d]
else cache-miss

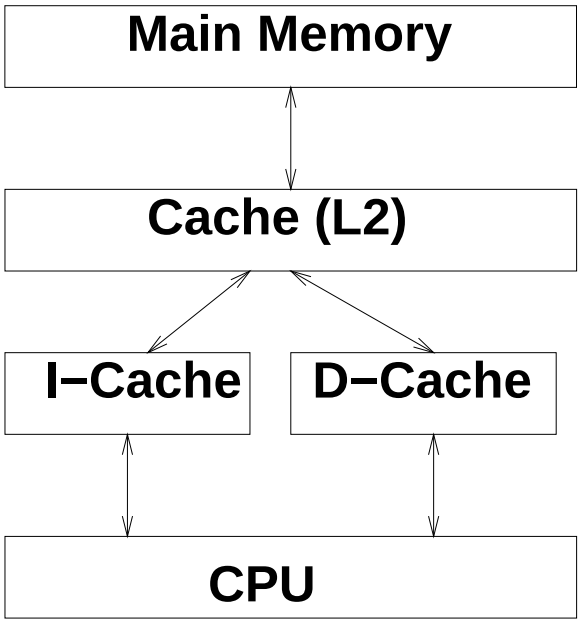


Cache Block Loading/Writing

- Read through
- Critical word first
- Write through (write merging)
- Write back
- Write allocate/no-allocate



Unified or Split Caches



Virtual Memory

Addresses used by programs *do not* correspond to actual addresses of the program/data locations in main memory. Instead, there exists a translation mechanism between the CPU and memory that actually associates CPU addresses (virtual) with their actual address (physical) in memory.

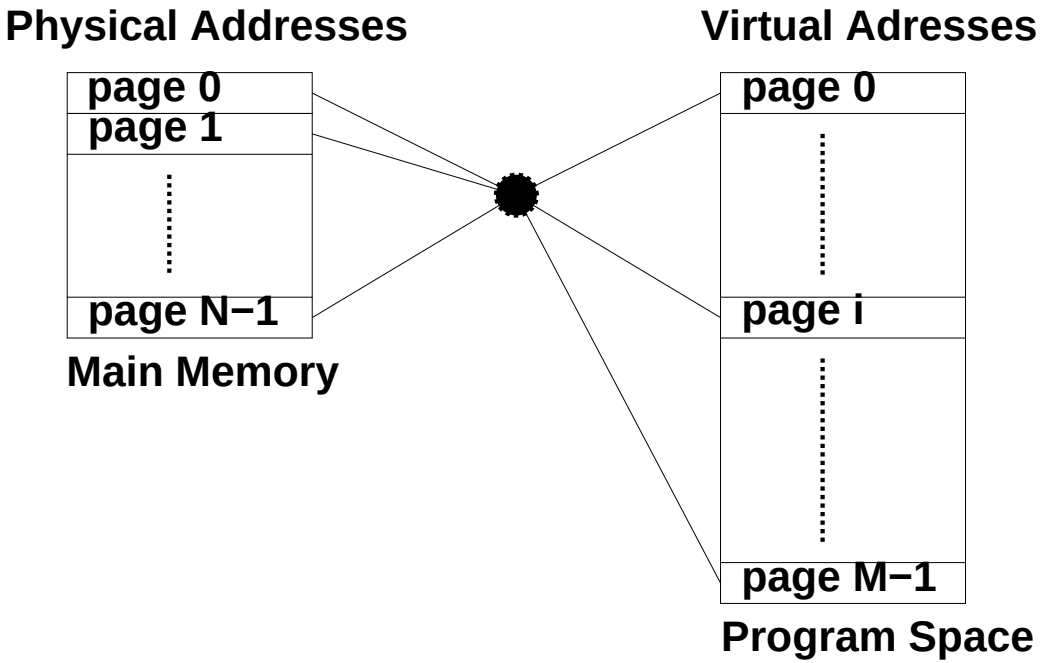
Two most important types:

- Paged virtual memory
- Segmented virtual memory

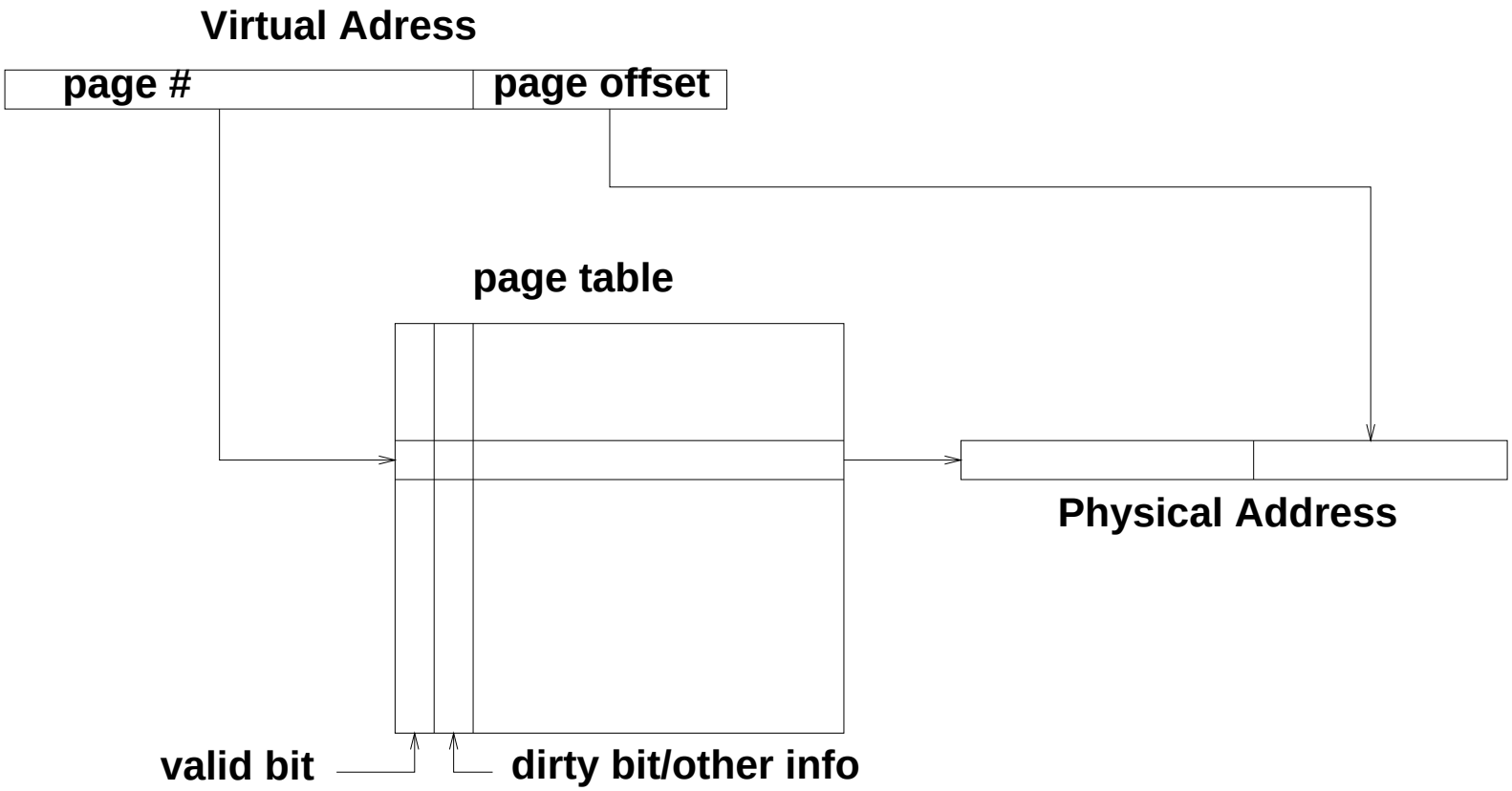
Demand Paged Virtual Memory

- Logically subdivide virtual and physical spaces into fixed sized units called *pages*.
- Keep virtual space on disk (swap for working/dirty pages).
- As referenced, bring pages into main memory (updating page table).
- Need page replacement algorithm: random, FIFO, LRU, LFU

Paged Virtual Memory: Structure



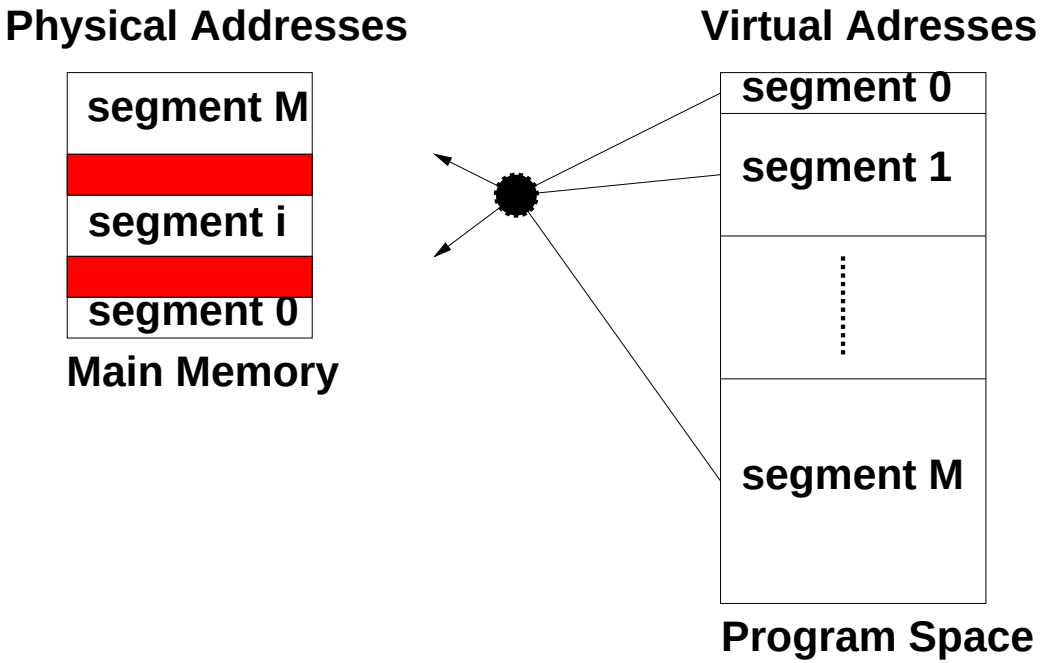
Paged Virtual Memory: Address Translation



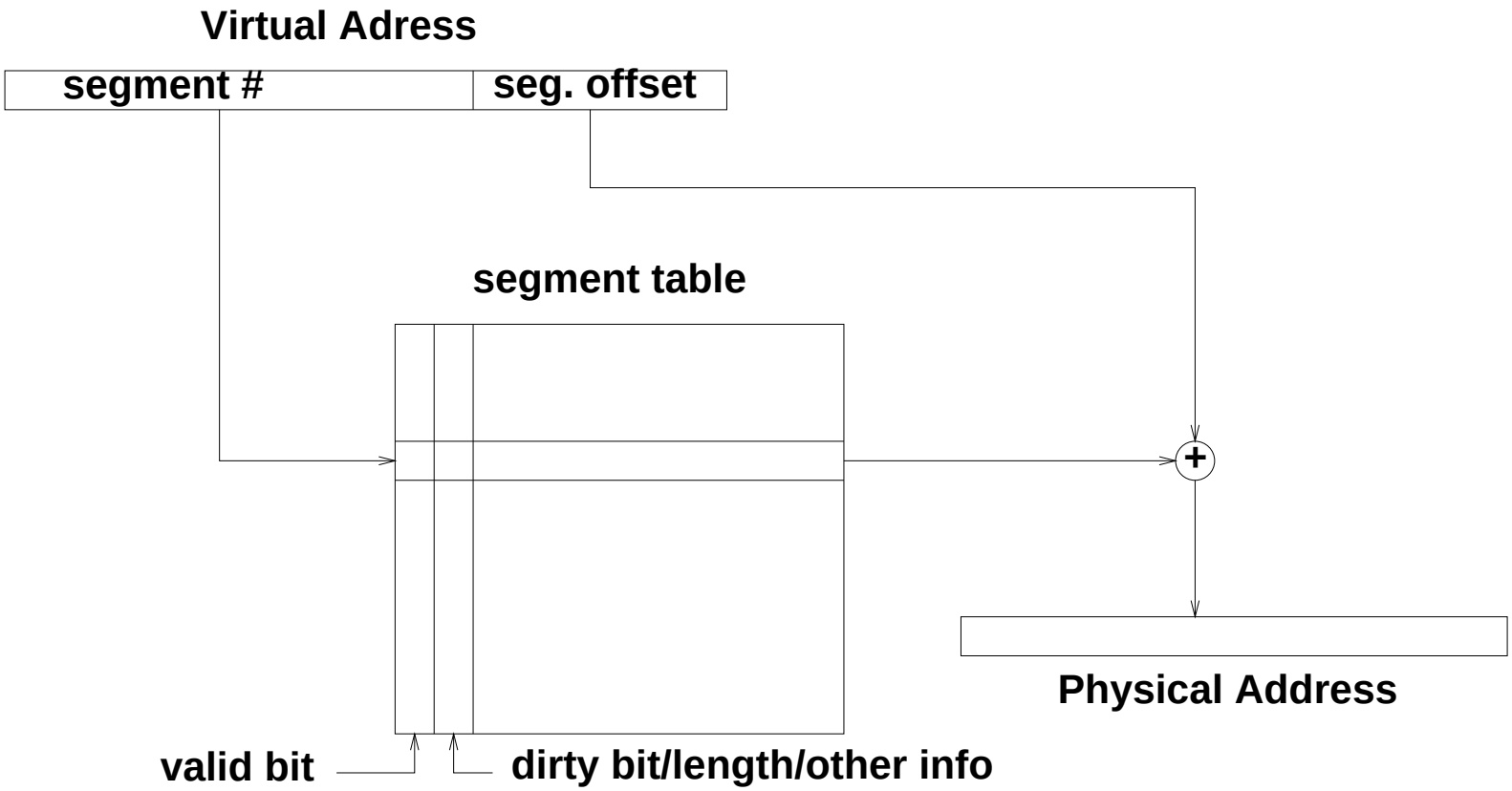
Segmented Virtual Memory

- Organize segments in virtual space by logical structure of program.
- Dynamically build segments in physical space (main memory) as segments are referenced.
- Keep virtual space on disk (swap for working/dirty pages).
- As referenced, bring segments into main memory (updating segment table).
- Need segment placement algorithm: best fit, worst fit, first fit.
- Need segment replacement algorithm.

Segmented Virtual Memory: Structure



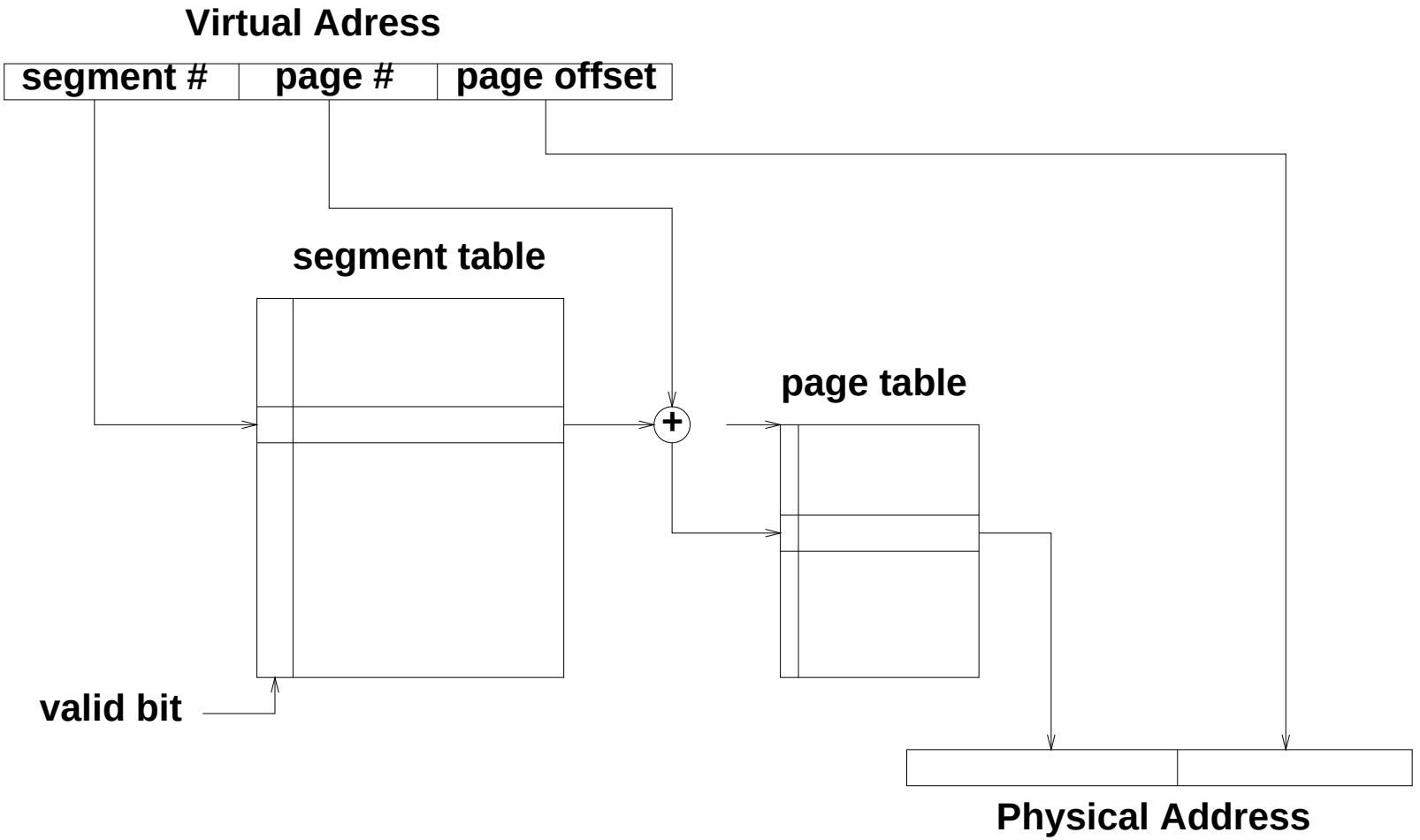
Segmented Virtual Memory: Address Translation



Combining Paged & Segmented Virtual Memory

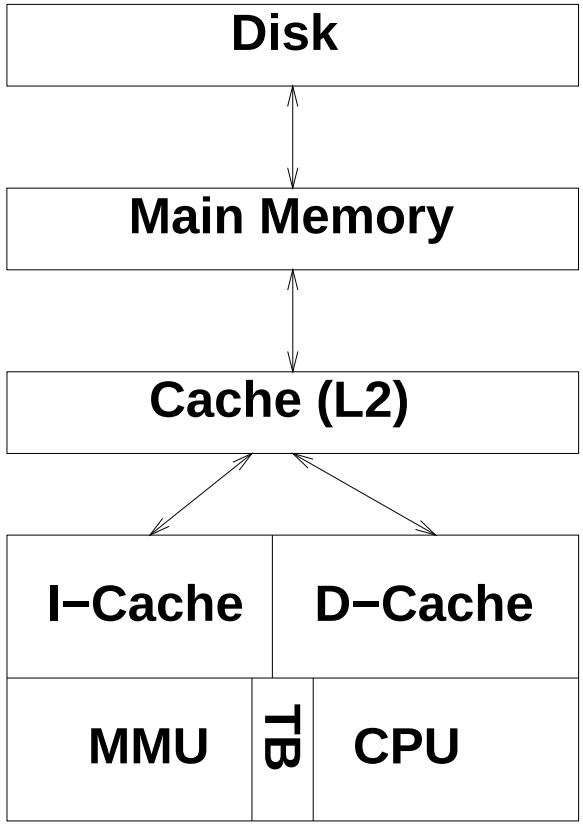
- linear segmentation
- name space segmentation

Segmented-Paged Virtual Memory: Address Translation



The Memory Hierarchy: Who does what to whom

- MMU: Memory management unit
- TB: Translation buffer
- H/W: search to main memory
- O/S: handles page faults (invoking DMA operation)
  - if dirty page, copy out first
  - move new to main memory (DMA)
- O/S: context swaps on page fault
- DMA: operates concurrently with other tasks



Memory Mangagement Unit

- Manages memory subsystems to main memory
- Translates addresses, searches caches, migrates data (to/from main memory out/in)

Translation Buffer

- Small cache to assist virtual $\rightarrow$ physical address translation process
- generally small (*e.g.,* 64 entries), size does not need to correspond to cache sizes

Satisfying A Memory Request

- L1 & L2 use physical addresses
- paged virtual memory
- ignoring details of TB misses

Satisfying A Memory Request

Satisfied in L1 cache:

1. MMU: translate address
2. MMU: search I or D cache as indicated by CPU, success (sometimes simultaneously with translation)
3. MMU: read/write information to/from CPU

Satisfying A Memory Request

Satisfied in L2 cache:

1. MMU: translate address
2. MMU: search I or D cache as indicated by CPU, failure
3. MMU: search L2 cache, success
4. MMU: move information between L1 & L2, critical word first?
5. MMU: read/write information to/from CPU

Satisfying A Memory Request

Satisfied in main memory:

1. MMU: translate address
2. MMU: search I or D cache as indicated by CPU, failure
3. MMU: search L2 cache, failure
4. MMU: move information between memory & L2
5. MMU: move information between L1 & L2, critical word first?
6. MMU: read/write information to/from CPU

Satisfying A Memory Request

Not in main memory:

1. MMU: translate address, failure trap to O/S
2. O/S: page fault, block task for page fault
   - O/S: if page dirty
     - O/S: initiate DMA transfer to copy page to swap
     - O/S: block task for DMA interrupt
     - O/S: invoke task scheduler
     - O/S: on interrupt continue
   - O/S: initiate DMA transfer to copy page to main memory
   - O/S: block task for DMA interrupt
   - O/S: invoke task scheduler
   - O/S: on interrupt:
     - update page table
     - return task to ready to run list

Translation Buffer Misses?

- Nothing special, it is a cache just like any other cache
- Generally fully associative

Tidbits

- Some pages must be pinned into main memory